第6次作业

## 第5章习题 P158

5.10证明：由二叉树的先序序列和中序序列能唯一确定一棵二叉树，并分别由下面的两个序列构造出相应的二叉树：

①先序：ABCDEFGHI ②先序：ABCDEFGHIJ

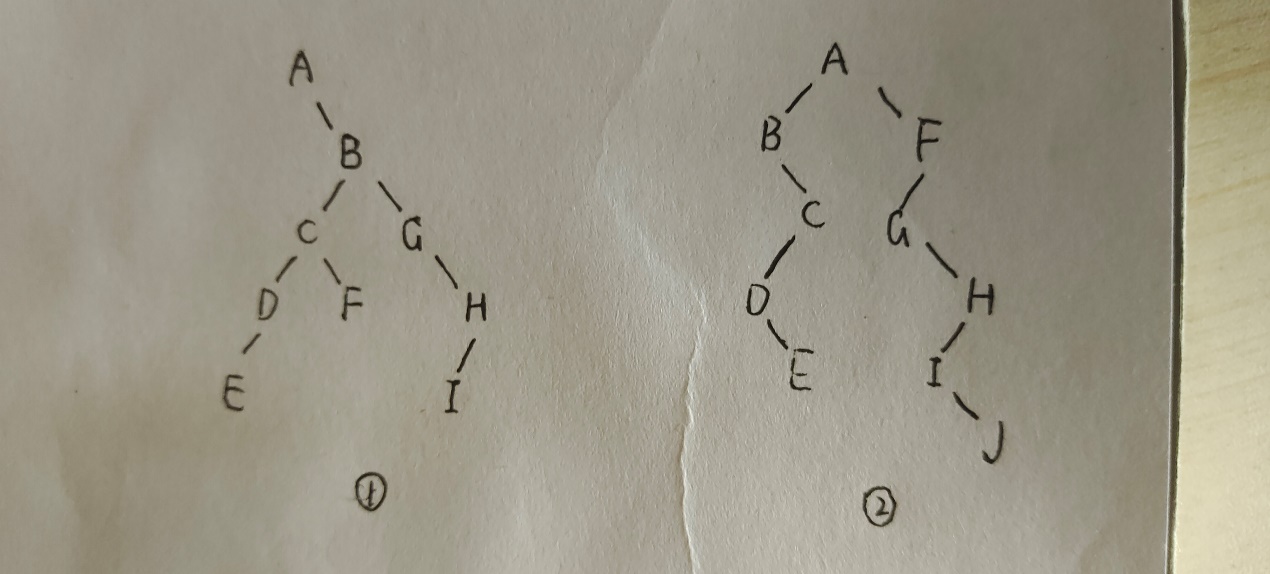
中序：ADECFBGIH 中序：BDECAGIJHF

证明：如果有一棵二叉树T，它的先序遍历序列为preT[]，中序遍历序列为inT[]，那么可以用以下步骤构建出这棵二叉树：

1. 从preT[]中取出第一个元素作为根节点。
2. 在inT[]中找到根节点的位置，将inT[]分成左子树的中序遍历序列inLeft[]和右子树的中序遍历序列inRight[]。
3. 根据inLeft[]中元素个数构造preT[]中左子树的先序遍历序列preLeft[]和右子树的先序遍历序列preRight[]，即从preT[1]开始，连续取相应数量的元素。
4. 对左子树重复步骤1-3，对右子树重复步骤1-3。左、右子树采用递归的方式构造。

通过上述步骤可以唯一地构建出一棵二叉树，并且任何一棵二叉树的先序序列和中序序列都能按照上述方法还原成唯一一棵二叉树，因此由二叉树的先序序列和中序序列可以确定唯一的二叉树。

答：



5.11证明：由二叉树的后序序列和中序序列能唯一确定一棵二叉树，并分别由下面的两个序列构造出相应的二叉树：

①后序：DCFEBIHGA ②后序：DECBGIHFA

中序：DCBFEAGHI 中序：DCEBAFHGI

证明：首先，当二叉树只有一个节点时，显然可以由其后序序列和中序序列唯一确定该二叉树。

接下来，假设对于任意n个节点的二叉树，都可以由其后序序列和中序序列唯一确定该二叉树。

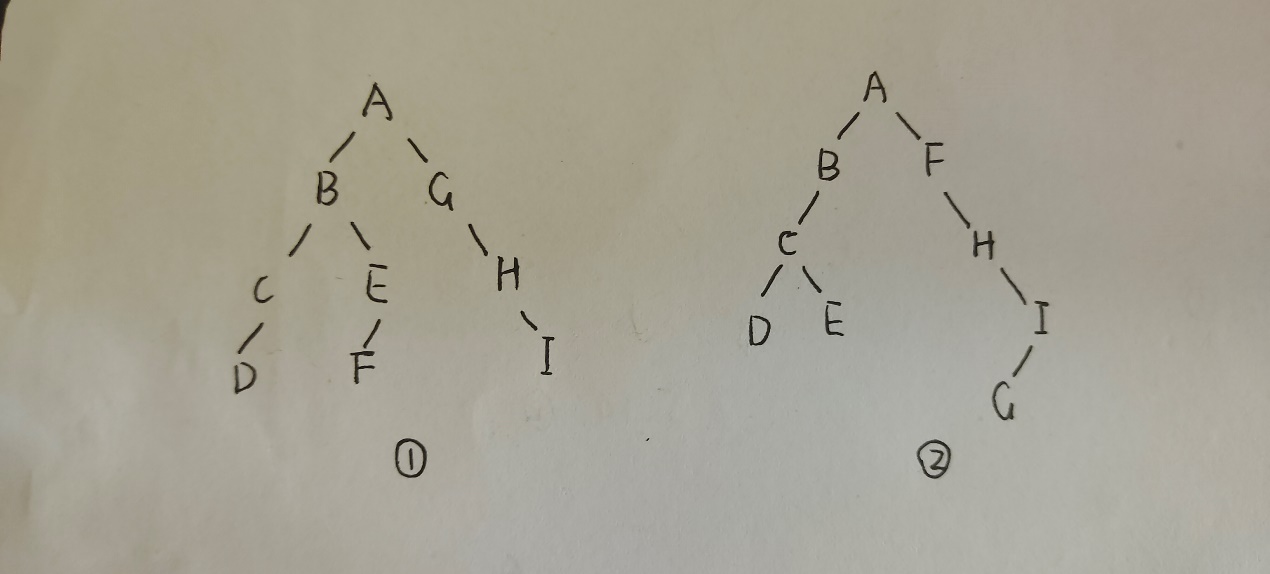
现在考虑一个n+1个节点的二叉树。根据后序序列的定义，最后一个节点是根节点。根据中序序列的定义，根节点将中序序列分成了左子树和右子树两部分。因此，可以根据中序序列找到根节点，并进一步确定左右子树各自的中序序列。

接着，可以根据右子树的大小，在后序序列中找到右子树的起始点和结束点，并进一步确定左右子树各自的后序序列。

由于左右子树分别包含n1和n2个节点，且n1 + n2 = n，根据归纳假设，可以唯一确定左右子树。因此，也可以唯一确定整个二叉树。

综上所述，由二叉树的后序序列和中序序列可以唯一确定该二叉树。

答：



5.12已知一棵二叉树的先序、中序和后序序列如下，其中各有一部分未给出其值，请构造出该二叉树。

先序：A\_CDEF\_H\_J

中序：C\_EDA\_GFI\_

后序：C\_ \_BHGJI\_ \_

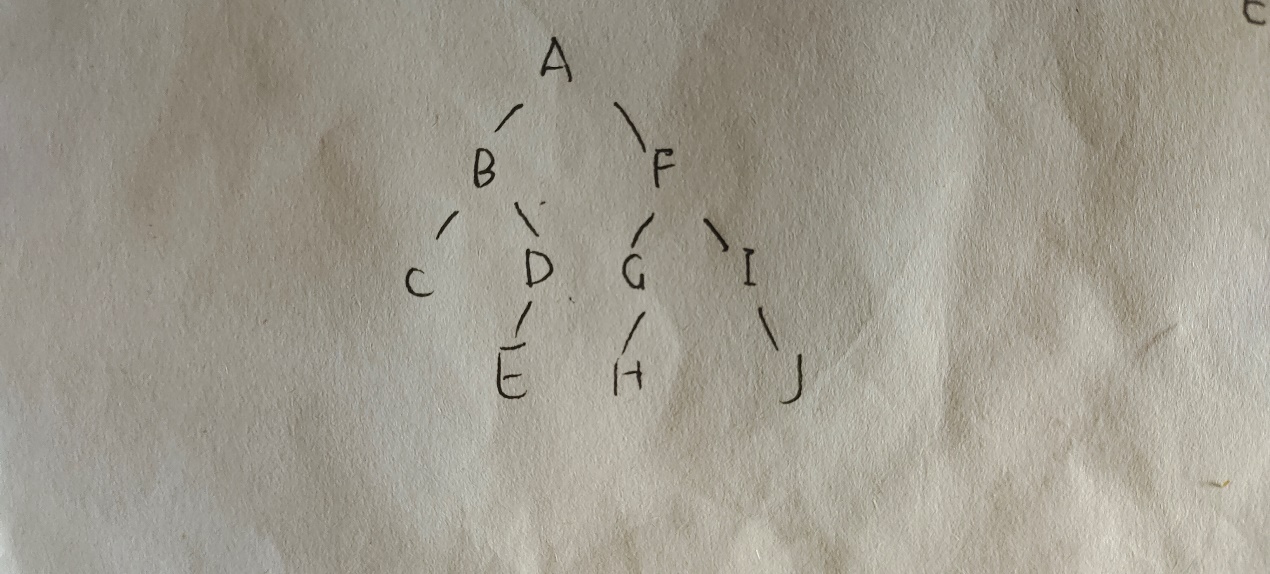
答：

补全后的遍历序列为：

先序序列：ABCDEFGHIJ

中序遍历：CBEDAHGFIJ

后序遍历：CEDBHGJIFA



5.16设计算法以输出每个结点到根结点之间的路径上的所有结点的值。

答：vector<int> findPath(TreeNode\* root, TreeNode\* target) {

if (!root) return {};

*// 如果当前结点是目标结点，则返回包含其自身的向量*

if (root == target) return {root->val};

*// 递归查找左子树和右子树*

auto left\_path = findPath(root->left, target);

auto right\_path = findPath(root->right, target);

*// 如果左子树和右子树均未找到目标结点，则返回空向量*

if (left\_path.empty() && right\_path.empty()) return {};

*// 将当前结点插入到路径中，并返回路径*

auto path = left\_path.empty() ? right\_path : left\_path;

path.insert(path.begin(), root->val);

return path;

}

5.18设计算法将一棵以顺序存储方式存储在数组A[]中的二叉树（已转换为完全二叉树，补充的虚拟结点为值为’#’）转换为二叉链表存储形式。

答：struct TreeNode {

char val;

TreeNode\* left;

TreeNode\* right;

TreeNode(char x) : val(x), left(nullptr), right(nullptr) {}

};

TreeNode\* arrayToTree(vector<char>& A) {

int n = A.size();

if (n == 0 || A[0] == '#') return nullptr;

*// 定义队列和根节点*

queue<TreeNode\*> q;

auto root = new TreeNode(A[0]);

q.push(root);

*// 创建二叉树*

for (int i = 1; i < n; ++i) {

if (A[i] == '#') continue;

*// 找到当前结点的父亲结点*

auto node = new TreeNode(A[i]);

auto p = q.front(); q.pop();

if (i % 2 == 0) p->right = node;

else p->left = node;

*// 将新结点插入到队列中*

q.push(node);

}

return root;

}

5.19分别设计出先序、中序和后序遍历二叉树的非递归算法。

答：struct TreeNode {

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

};

*// 先序遍历*

vector<int> preorderTraversal(TreeNode\* root) {

vector<int> res;

stack<TreeNode\*> st;

if (root) st.push(root);

while (!st.empty()) {

auto node = st.top(); st.pop();

res.push\_back(node->val);

if (node->right) st.push(node->right);

if (node->left) st.push(node->left);

}

return res;

}

*// 中序遍历*

vector<int> inorderTraversal(TreeNode\* root) {

vector<int> res;

stack<TreeNode\*> st;

auto node = root;

while (node || !st.empty()) {

if (node) {

st.push(node);

node = node->left;

} else {

node = st.top(); st.pop();

res.push\_back(node->val);

node = node->right;

}

}

return res;

}

*// 后序遍历*

vector<int> postorderTraversal(TreeNode\* root) {

vector<int> res;

stack<TreeNode\*> st;

auto last = static\_cast<TreeNode\*>(nullptr);

while (root || !st.empty()) {

if (root) {

st.push(root);

root = root->left;

} else {

auto node = st.top();

if (node->right && node->right != last) {

root = node->right;

} else {

st.pop();

res.push\_back(node->val);

last = node;

}

}

}

return res;

}